函数

定义函数

def func_name([arg1 [, arg2, ... argN]]):
    func_body

- 形参： 函数定义时声明的参数。
- 实参： 函数调用时传入的参数。
- 函数只需要定义一次，就可以被多次使用。
- 当函数被调用时，才执行函数体，定义时不执行。

```
def plus(num):
    print(num + 1)


""" 调用函数 """
plus(2)  # 3
plus(5)  # 6

f = plus
print(plus)
print(f)
f(2)  # 3
f(5)  # 6
```

return 用法

- 把后面跟着的对象返回给函数调用方，并结束所在的函数
- return 后面可以跟一个对象，多个对象，甚至不跟任何对象
- return 后面什么都不跟，等价于 return None
- 函数执行时，没有遇到 return，也等价于 return None

```
# 返回一个对象
def add1(left, right):
    res = left + right
    return res


def add2(left, right):
    return left + right


# 返回多个对象，自动打包成一个元组
def add3(left, right):
    res1 = left + right
    res2 = left * right
    return res1, res2


def add4(left, right):
    return left + right, left * right


# return None
```

```
def add5(left, right):
    print(left + right)
    return


# return None
def add6(left, right):
    pass


print(add1(3, 4))
print(add2(3, 4))
print(add3(3, 4))
print(add4(3, 4))
print(add5(3, 4))
print(add6(3, 4))
```

参数传递

- 传不可变对象 & 传可变对象

```
def func(b):
    print(id(a), a)
    print(id(b), b)


a = 789
```

```
func(a)


def func(b):
    print(id(a), a)
    print(id(b), b)


a = [789]
func(a)


def func(b):
    b.append(345)


a = [789]
func(a)
print(a)
```

参数分类

必需参数

- 必须接收一个实参的形参，多了少了都不行

```
def func(a, b):
    print(a - b)


func(3, 4)
func(3, b=4)
func(a=3, b=4)
```

位置参数

- 按照从左往右的顺序将实参传递给对应的形参

```
def func(a, b):
    print(a - b)


func(3, 4)  # -1
func(4, 3)  # 1
```

关键字参数

- 按照指定的名称将实参传递给对应的形参，与位置顺序无关
- 关键字参数必须放在位置参数的后面

```
def func(a, b):
    print(a - b)



func(a=3, b=4)  # -1
func(b=4, a=3)  # -1
func(3, b=4)  # -1
```

默认参数

- 有接收到实参，使用实参，没有接收到实参时，才会使用默认值

```
def func(a, b=4):
    print(a - b)



func(3)  # -1
func(3, 5)  # -2
```

不定长参数

- *args：接收[0, +∞)个位置参数，贪婪的，将它们打包成一个元组，如果没有接收到实参，则为空元组。
- **kwargs：接收[0, +∞)个关键字参数，贪婪的，将它们打包成一个字典，如果没有接收到实参，则为空字典。必须放在所有形参的最后。

```
def func(*args):

```

```
    print(args)


func()
func(3, 1, 4, 6)


def func(**kwargs):
    print(kwargs)


func()
func(a=3, b=2, c=4)


def func(*args, **kwargs):
    print(args)
    print(kwargs)


func()
func(1, 2, a=3, b=4)
```

特殊参数

- 默认情况下，实参的传递形式可以是位置参数或关键字参数
- 可以用 `/` 和 `*` 来限制参数的传递形式
- 其中 `/` 为仅限位置参数，限制在它之前的形参只能接收位置参数

- 其中 `*` 为仅限关键字参数，限制在它之后的形参只能接收关键字参数
- 这两个特殊参数只是为了限制参数的传递形式，不需要为它们传入实参

```
def func(pos1, pos2, /, pos_or_kwd, *, kwd1,
kwd2):
    pass


func(1, 2, 3, kwd1=4, kwd2=5)
func(1, 2, pos_or_kwd=3, kwd1=4, kwd2=5)
```

匿名函数

- 格式： lambda [arg1 [, arg2, ... argN]] : expression
- 匿名函数的参数可以有多个，但是后面的 expression 只能有一个
- 匿名函数返回值就是 expression 的结果，而不需要使用 return
- 匿名函数可以在需要函数对象的地方使用（如：赋值给变量、作为参数传入其他函数等），因为匿名函数可以作为一个表达式，而不是一个结构化的代码块

```
print((lambda: 'It just returns a string')())
f = lambda: 'It just returns a string'
print(f())


(lambda x, y, z: print(x + y + z))(1, 2, 3)
f = lambda x, y, z: print(x + y + z)
f(1, 2, 3)


tup = (8, 5, -9, 6, 2)
print(sorted(tup, key=lambda x: -x if x < 0 else
x))
```

封包、解包

封包

- 将多个值同时赋值给一个变量时，会自动将这些值打包成一个元组

```
tup = 345, 'hello', 789
print(tup)
```

解包

解包是针对可迭代对象的操作

- 赋值过程中的解包

```
a, b, c = [4, 3, 'a']
print(a)  # 4
print(b)  # 3
print(c)  # 'a'

a, *b, c = 'hello'
print(a)  # 'h'
print(b)  # ['e', 'l', 'l']
print(c)  # 'o'

a, *b, c = 'he'
print(a)  # 'h'
print(b)  # []
print(c)  # 'e'

*a, = 'hel'
print(a)  # ['h', 'e', 'l']
```

```
_, *b, _ = [4, 3, 5, 7]
print(b)  # [3, 5]
```

- 在可迭代对象前面加一个星号（*），在字典对象前面加双星（**），这种解包方式主要运用在函数传参的过程中

```
def func(a, b, c):
    print(a, b, c)


"""
在函数传实参时, *iterable可以将
该iterable解包成位置参数
"""
tup = (1, 2, 3)
func(*tup)  # 等价于func(1, 2, 3)

d = {'a': 1, 'b': 2, 'c': 3}
func(*d)  # 等价于func('a', 'b', 'c')


"""
在函数传参时, **dict可以将
该dict解包成关键字参数
"""
func(**d)  # 等价于func(a=1, b=2, c=3)
```

命名空间与作用域

命名空间

定义：命名空间（Namespace）是一个从名称到对象的映射

实现：大部分命名空间当前由 Python 字典实现（内置命名空间由 builtins 模块实现）

作用：提供了在项目中避免名字冲突的一种方法（各个命名空间是独立的，在一个命名空间中不能有重名，但不同的命名空间是可以重名而没有任何影响的）

内置命名空间

- 包含了所有 Python 内置对象的名称
- 在解释器启动时创建，持续到解释器终止

全局命名空间

- 包含了模块中定义的名称，如：变量名、函数名、类名、其它导入的名称
- 在模块被读入时创建，持续到解释器终止

局部命名空间

- 包含了函数中定义的名称，如：函数中的变量名、参数名
- 在函数被调用时创建，持续到该函数结束为止

```
import this


def func1(arg1, arg2):
    num = 666
    print(locals())  # 返回局部命名空间


def func2(arg1, arg2):
    num = 777
    print(locals())


num = 111
func1(222, 333)
func2(444, 555)

print(globals())  # 返回全局命名空间
# 在全局作用域，locals()等价于globals()
print(locals())
```

命名空间查找顺序

- 局部命名空间 >> 全局命名空间 >> 内置命名空间

作用域

定义：Python 程序可以直接访问命名空间的正文区域

作用：决定了哪一部分区域可以访问哪个特定的名称

分类：（L - E - G - B 作用域依次增大）

- 局部作用域（Local） - L
- 闭包函数外的函数中（Enclosing） - E
- 全局作用域（Global） - G
- 内置作用域（Built-in） - B

规则：在当前作用域如果找不到对应名称，则去更大一级作用域去找，直到最后找不到就会报错

注意：模块、类以及函数会引入新的作用域，而条件语句，循环语句并不会

局部作用域

```
def func(x, y):
    """
    函数内部区域可以直接访问该函数所对应的局部命名空间,
    所以该区域为 局部作用域(Local)
    """
    a = 3
    b = 4
    print(x, y, a, b)


func(1, 2)
```

闭包函数外的函数中

```
def outer(a):
    """
    在inner函数的外部且在outer函数的内部区域,
    可以直接访问outer所对应的局部命名空间,
    所以该区域为 闭包函数外的函数中(Enclosing)
    """
    b = 2

    def inner(c):
        """ 局部作用域 """
        return a + b + c

    return inner
```

```
""" 全局作用域 """
print(outer(1)(3))
```

全局作用域

```
def func():
    pass


"""
函数外部区域可以直接访问该模块所对应的全局命名空间，
所以该区域为 全局作用域(Global)
"""
a = 3
b = 4
print(a, b)
```

内置作用域

- 能够直接访问内置命名空间的正文区域为内置作用域
- builtins模块的全局作用域，相当于Python的内置作用域

global 和 nonlocal

- 当内部作用域想要给外部作用域的变量重新赋值时，可以用 global 或 nonlocal 关键字

```
def outer():
    global a, b
    a, b, c, d = 3, 4, 5, 6
    print(a, b)

    def inner():
        global a, b
        nonlocal c, d
        a, b, c, d = 7, 8, 9, 0

    inner()
    print(c, d)



a, b = 1, 2
outer()
print(a, b)
```